Using Visualization to Foster Object-Oriented Program Understanding

Dean F. Jerding and John T. Stasko

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
E-mail: {dfj,stasko}@cc.gatech.edu

Technical Report GIT-GVU-94-33
July 1994

Abstract

Software development and maintenance tasks rely on and can benefit from an increased level of program understanding. Object-oriented programming languages provide features which facilitate software maintenance, yet the same features often make object-oriented programs more difficult to understand. We support the use of *program visualization* techniques to foster object-oriented program comprehension. This paper identifies ways that visualization can increase program understanding, and presents a means for characterizing both static and dynamic aspects of an object-oriented program. We then describe the implementation of a prototypical tool for visualizing the execution of C++ programs. Based on this work, we define a framework for the visualization of object-oriented software which requires little or no programmer intervention and provides a mechanism which allows users to focus quickly on particular aspects of the program.

Keywords: program visualization, program comprehension.

1 Introduction

The object-oriented programming paradigm has developed partly in response to the failure of other programming languages to provide adequate maintenance facilities. Features of object-oriented design such as encapsulation, polymorphism, and re-use through inheritance can help support software maintenance tasks. However, the potential benefits provided by object-oriented programming during the maintenance process do not come without cost. For example, object-oriented programmers must *understand* inheritance, dynamic binding, and various forms of polymorphism. A software development environment that purports to assist object-oriented program understanding, therefore, must address these issues–many of which arise at run-time.

Typically, a programmer develops a computer program to solve a problem. As a first step, the programmer draws upon knowledge of the problem domain to create a mental model of a possible solution. Using programming domain knowledge, this idea is then mapped into a conceptual model of a computer program which will solve the problem. The conceptual model (in the programming domain) is then realized using a programming language. When the resulting program needs to be modified or enhanced, some or all of these mappings may be misinterpreted. The problem of program understanding thus becomes one of the most important tasks in software re-use and maintenance.

There are many theories as to how program comprehension is performed. Brooks originally proposed the idea that comprehension is essentially a maintainer reconstructing the mappings that the programmer originally created [Bro83]. The program understanding process may proceed in a *bottom-up*[BM82, Bro83, Pen87] or *top-down*[SAE88] manner, or a combination of both[Let86, vMV93], depending on the available cues, the type of maintenance[vMV93], and the maintainer's syntactic and semantic knowledge base[YB93]. The process might also be *systematic*–an attempt to understand the entire program, or *as-needed*, where only the parts of a program necessary to carry out a particular task are investigated[LPLS86].

Another view holds that program understanding takes place in a *feedback loop*[DPKV94] where the program implementation is compared to the maintainer's conceptual model of how the program should solve the problem. Research in the area of program comprehension has been focused on providing tools which make this feedback loop more effective. Two main approaches have been followed here: 1) to allow maintainers access to knowledge about the design of the program, as described by Younger and Bennett[YB93], or 2) to provide an analysis of the many inter-dependencies existing in a program, including data-flow and control-flow, as in[LA93]. Both of these approaches are often based upon creating some internal representation of a program from which information can be extracted and presented to the user. Some tools have begun to present this information graphically, such as the CARE environment[LAD+93] and FIELD[Rei90].

We support a different approach to program understanding: *program visualization*. Program visualization is a sub-set of the area known as *software visualization*–the use of graphics and animation to visually describe and illustrate software and its function[PBS93, SP92]. In program visualization, the medium being visualized is a computer program. The basic premise of visualization is that users can better understand and investigate the inner-workings of their software by seeing it portrayed visually.

We believe that the object-oriented programming paradigm is an especially natural

foundation for visualization because it fundamentally involves the manipulation of concrete "things": instances, messages, methods, and so on. Undoubtedly, programmers will already have a mental model of their software in which these entities have visual manifestations. Building visualization tools for object-oriented systems follows naturally from their correspondence to a visual representation.

This paper describes our efforts to identify appropriate, informative, and easily comprehensible visualizations that communicate what programmers and maintainers want and need to know about their software. We begin by identifying in Section 2 the most problematic features of object-oriented software development and maintenance, problems that visualization might help solve. Section 3 discusses a means through which object-oriented programs can be characterized. Drawing from our previous work on visual design of object-oriented programs, we have developed a prototype system for visualizing object-oriented program execution, which is described in Section 4. Based on those results, we have defined a visualization framework that can effectively collect and present useful views of object-oriented programs. Section 5 defines the goals of this framework and Section 6 describes the framework itself. The final two sections discuss related work and future directions of our own research, respectively.

2 Object-Oriented Software Development Challenges

The advent of object-oriented programming languages has provided software developers with new tools. Programmers have adopted the object-oriented approach in order to take advantage of better data abstraction, improved modularity, function overloading, and code re-use. These characteristics can potentially benefit both software development and software maintenance.

The object-oriented paradigm is somewhat of a double-edged sword, however. The powerful features of inheritance, dynamic binding, and polymorphism also make object-oriented programs harder to understand and maintain. Class descriptions are often distributed among several files, and function overloading is common. Additionally, static code traces cannot fully describe the execution of object-oriented programs.

Meyers gives a detailed example of a common, "representative" problem often encountered in object-oriented programming and debugging[Mey90]. The task involves locating the function body that would be invoked from a particular function call in a C++ application. In a traditional procedural language, this might simply involve looking for the corresponding function declaration in successive enclosing scopes. However, in this example the problems of inheritance and dynamic binding force the programmer to examine nine classes (including duplicates), which may be spread out over multiple files. Meyers goes on to point out how this simple task of tracing function calls is fundamental in making bug fixes, performing code tuning, and adding enhancements to existing software.

In order to take full advantage of the object-oriented paradigm, software developers must be provided with tools that alleviate such side-effects. Wilde and Huitt have identified several areas in which tool support is needed: tracing dynamic message binding, analyzing dependencies between classes, understanding high-level aspects of a system, locating system functionality, and resolving polymorphism[WH92].

A tool that tracks the dynamic message handling by objects during program execution

would be able to provide users with dynamic binding information. Method invocation during message handling is an example of a more global aspect of object-oriented programs: object dependencies. Such dependencies might be class-to-class, class-to-method, or method-to-message. Classes whose methods are invoked often are clearly candidates for optimization. Analyzing these dependencies can be complicated by dynamic binding and polymorphism. Additionally, displaying the multi-dimensional nature of the dependencies would be difficult with purely textual screens[WH92].

In the object-oriented software development process, the inheritance hierarchy is often a dynamic entity that evolves over time. Part of this evolution involves moving class methods up or down in the hierarchy, to meet implementation constraints. If class dependency information were available, this process could be performed more efficiently.

Another phenomenon related to dynamic binding occurs when method calls are circularly made to different classes in a single object's inheritance hierarchy. This occurrence was recognized by Taenzer, Ganti, and Podar as the yo-yo effect[TGP89]. If a programmer had access to a visualization of the method binding, such a phenomenon could be easily identified.

Maintenance of software systems is often complicated by the transiency of programmers in software development organizations. It is often difficult for a new programmer to understand the complex relationships between objects present in large object-oriented systems[LMR92]. Again, program understanding is complicated by dynamic binding and inheritance hierarchies. One suggestion for increasing understanding might be a graphical view of the inheritance hierarchy[WH92], or a graphical visualization of the communication between objects in the system.

Another maintenance problem arises when the naming of messages does not correspond consistently to the action taken by objects receiving the messages. This problem is particularly important in object-oriented systems, where function overloading is both permissible and common[WH92]. Wilde and Huitt go on to point out that, "graphic displays of object relationships would seem to be very useful." (page 1043)

It is clear that the object-oriented approach to software development includes pitfalls for program developers and maintainers. We believe that software visualization tools would be invaluable toward reducing programmer effort, especially during the enhancement and maintenance phases of software development. Such tools aid in understanding object-oriented systems during maintenance, and in lower-level debugging tasks during object-oriented program development.

3 Characterizing Object-Oriented Programs

Given that tools which support visualization of object-oriented programs would be useful, the first question is *What needs to be visualized?* In other words, what entities, relationships, and actions exist in the program that might be portrayed visually? In this section we describe a means through which an object-oriented program can be characterized, thereby providing a basis upon which the visualizations can be built.

Clearly, object instances are the basic entities in an object-oriented program. The inheritance relationship is specified by the class hierarchy. Important actions which characterize program execution include object creation, object deletion, and message passing. These

Event	Parameters
Class Define	class name
Parent Define	class name, parent name
Method Define	class name, method name, member type, member location, method type, return type, argument list
Attribute Define	class name, attribute name, member type, member location, type
Friend Define	class name, friend name
Global Function Define	function name, return type, argument list
Instance Create	timestamp, class name, instance name, this ptr, filename, line number
Instance Destroy	timestamp, class name, instance name, this ptr, filename, line number
Constructor Invoke	timestamp, class name, this ptr, filename, line number, argument list
Constructor Return	timestamp, class name, this ptr, filename, line number
Destructor Invoke	timestamp, class name, this ptr, filename, line number
Destructor Return	timestamp, class name, this ptr, filename, line number
Method Invoke	timestamp, class name, method name, this ptr, filename, line number, argument list
Method Return	timestamp, class name, method name, this ptr, filename, line number, argument
Global Function Invoke	timestamp, function name, filename, line number, argument list
Global Function Return	timestamp, function name, filename, line number, argument

Table 1: Object-Oriented Program Events

actions are *interesting events* which need to be monitored during program execution.

One common method for creating program visualizations is to generate traces of interesting events which describe the execution of the program. To describe object-oriented programs, events should record both object creation and deletion. In most object-oriented languages, constructors (destructors) are called when an object is created (deleted). These invocations should also be traced.

Typically, an object handles a message by invoking the corresponding method within its inheritance hierarchy. Method invocations and returns thus need to be monitored. C++ is not a true object-oriented program in that it allows procedural execution of global functions. Thus, our event tracking mechanism should track global function invocations.

C++ also allows the definition and overloading of "operators", both within a class and globally. Common operators include arithmetic operators and stream operators. Operator invocation can be tracked automatically by considering local operators as methods and global operators as global functions.[1]

In order to describe the inheritance hierarchy (class hierarchy) of an object-oriented program, "static" definition events could exist to describe classes, including their methods and attributes. Definition events would also describe any global functions in the program. These events must be generated statically, prior to the actual execution of the program. Such static events could be processed by the visualization before the dynamic events.

A list of interesting events that we have identified to foster visualization of an object-oriented program's execution is shown in Table 1. This list is similar to the types of events

[1]This is semantically correct as defined by the C++ language.

Current focus	**Related entity**	**Entity's depiction** (visual attribute)
Class	Itself	Full color, bold outline
	Base class(es)	Light color, inheritance connection arrows
	Derived class(es)	Light color, inheritance connection arrows
	Instance of class	Light color, bold outline
	Instance of a derived class	Light color
	Friend class or function	Hand icon
Instance	Itself	Full color, bold outline
	Its class	Light color, bold outline
	Classes it inherits from	Light color
	Visible instances	Double, broken outline
	Subordinate instances	close proximity
Function	Itself	Full size, bold outline

Table 2: Graphical encoding of relationships to a current focus program entity.

tracked by [DPHKV93], as both sets of events seek to characterize object-oriented program execution at a conceptual level. Although the semantics of C++ primarily guided the development of this list, the events were designed to be flexible enough to characterize other languages as well.

4 A Prototype for Visualization

In Section 3 we attempted to answer the question: *What needs to be visualized?* This section then addresses the question: *How do we visualize it?* Motivated by the realization that tools and methods for visualizing the execution of object-oriented software were clearly lacking, we first set out to construct a basic visualization of an executing object-oriented program.

Previous work by Shilling and Stasko involved constructing the GROOVE visual design tool[SS92]. The GROOVE tool is designed to help programmers visually specify both the static structure and the dynamic protocols of an object-oriented program. GROOVE's visual paradigm employs shape, color, and animation to portray objects and relationships. For example, classes are represented as upside-down triangles. Arrows from the bottom of a class to the top of another class represent inheritance. Instances are represented by ovals, and global functions and methods by rectangles. Message passing is shown by animating the drawing of arrows between instances.

GROOVE utilizes the concept of a *current focus* in the visual display. Any program entity in the display can become the current focus by user selection. During program visualization the instance last receiving a message becomes the focus. Once an entity becomes the focus, all other objects update their view to reflect their relationship to the focus entity. Encoded relationships include inheritance, friendship, visibility, class, etc. Table 2 shows a brief list of relationships and how they are encoded.

Based on the visual paradigm from GROOVE, we have developed a basic view of an executing program. The view contains three basic entities: a tree structure representing the class hierarchy, rectangular nodes representing global functions, and circular nodes representing instances. The view is intended to graphically animate the message-passing that occurs during execution of object-oriented programs.

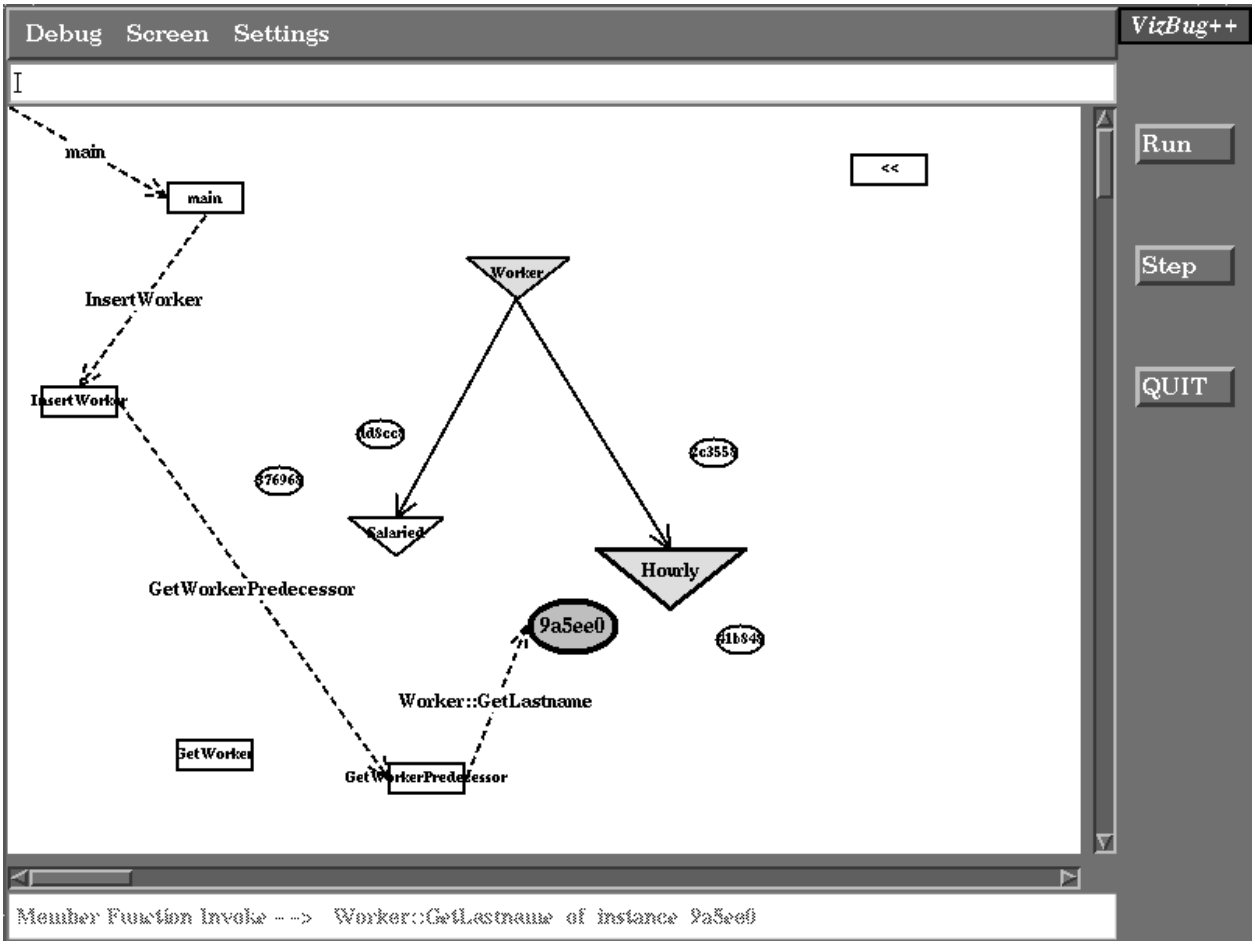

Figure 1: Simple View of an Executing O-O Program

A library of visualization functions exists to construct and animate the view appropriately for each type of event. Simple C++ programs can be hand-annotated with these calls at places where interesting events occur in the program. Execution of the annotated source program then produces calls into the visualization library and causes the view to be updated.

We call this system *VizBug++*, for visual debugging of C++ programs. A simple user interface provides both continuous and discrete (step mode) control of the event visualization. The resulting view provides a graphical animation of the various events taking place in the program. Figure 1 shows a sample display for a program with three classes and a very small number of instances.

Instance construction is displayed by instances "hatching" out of class nodes in the inheritance hierarchy, and moving to other class nodes for each successive constructor call. Color is used to encode inheritance attributes. Function invocations are animated by drawing arrows from the calling instance or global function to the callee. Arrows are labeled with the function name and class name, when appropriate. Function returns are displayed by retracting the arrow from the callee back to the caller. A simple layout algorithm is performed which places the inheritance hierarchy toward the center of the view, global functions toward the edges, and instances near their classes.

By stepping through the visualization of the program execution, users can observe existing objects and messages. User's can browse the relationships between existing entities by pointing and clicking. The resulting information can be useful toward understanding the dynamics of the program execution, thereby benefiting the program development and maintenance process.

From the results of this "prototypical" view and our experience implementing *VizBug++*, it is clear that gathering the necessary information to construct useful visualizations is indeed a difficult problem. View layout and information overload are major problems encountered in this simple view. Presenting the information in an organized and informative way seems to require multiple coordinated views, with different levels of abstraction. Given the role that object-oriented programming has begun to play in the software development arena, these problems merit further exploration.

5 Visualization Objectives

Based on existing object-oriented software development challenges and on insight gained from our prototype visualization system, we have set forth four main objectives that a framework for the visualization of object-oriented software must realize:

- **Little or no programmer intervention**. Generating the visualizations we develop, once integrated into a programming environment, should require little or no programmer intervention. The visualizations should be generated by something as simple as using a different compiler or filter program, or by adding an extra flag to a compilation. This implies that the visualizations must be driven from information that can be acquired by the underlying software prior to and during execution.

- **Present the "right" things**. The visualizations developed should present the most important aspects of a program or software system, those that will be most useful to programmers. The visualizations must convey information about the dynamic execution of a program and cumulative summary information about an entire execution as a whole. The visualizations and animations developed must address all the key challenges we identified in Section 2, and any other new attributes that we uncover as this research progresses.

- **Allow viewers to focus quickly**. The visualizations must be designed in a way that facilitates programmers focusing on their particular concerns in a timely and straightforward matter. For instance, suppose that the problematic scenario a programmer wishes to examine occurs well into a program's execution. The programmer should not have to take a long time to wade through impertinent animations to arrive at the point of interest. Conversely, users should be able to examine execution states that have already occurred, by "rewinding" execution. Additionally, visualizations should be constructed in a way that presents an overview of a program, but also in a way that supports simple navigation toward an attribute or feature of particular concern.

- **Handle real-world problems**. The visualizations developed should not be restricted to presenting only small, laboratory programs and systems. Rather, the visualizations must be applicable to large object-oriented systems perhaps involving hundreds of thousands of lines of code.

Objectives such as these have for quite a while been recognized as some of the most important open problems within all software visualization research, not just that involving object-oriented systems.

6 Defining the Visualization Framework

This section describes a framework for visualizing object-oriented software, based on the objectives set forth in the previous section. The entire visualization process takes as input the source code, and produces interactive graphical, animated *views* of the executing program.

The tasks required to construct these visualizations can be divided into two basic areas: *program event generation* and *program event visualization*. Event generation involves tracing particular events during the execution of an object-oriented program, such as object creation, object deletion, and member function invocation. Once the events have been generated, program visualization can be performed based on the events. Accordingly, the *visualization framework* can be separated into two functional modules, one for event generation and the other for visualization of the program events.

In line with the four major goals presented in the previous section, our framework for visualizing object-oriented program execution must support several additional capabilities:

- Automatic extraction of important program events.
- Interactive or post-mortem visualization.
- Two-way navigation through the program event stream.
- Direct manipulation of view contents.
- Structured interaction between views.
- Easy addition of new views.

We propose a visualization framework as shown in Figure 2. The event generation phase would be performed by an **Event Generator** which utilizes an **Event Trace Library**, while the event visualization responsibilities would be divided between an **Event Manager** and a **Visualization Manager**. The remaining parts of this section outline the conceptual workings of the visualization framework, and describe how it will support the objectives we have established.

6.1 Event Generation

The event generation phase involves the generation of interesting program events to drive our visualizations. As discussed earlier, we have identified interesting events which describe the execution of object-oriented programs, such as object creation/destruction and method calls. Additionally, static information about the program is also needed, such as the structure of the inheritance hierarchy. In our framework, the Event Generator is responsible for these event generation tasks.

Our goal is to automate the event generation process such that it is unintrusive to the programmer. There are various degrees to which the generation process can be automated, ranging from reconstructing events based on dynamic trace/profile information, to modifying a standard compiler or the executable such that the information is generated as the

Figure 2: Visualization Framework

program runs. Our visualization framework will allow us to explore and evaluate several methods within this range.

6.2 Event Visualization

Event visualization involves creating graphical, animated views of program execution based on the collected events. The event visualization phase involves two different tasks, *managing* the collected program events and *visualizing* the events. We believe these tasks need to be separated in order to improve visualization performance and facilitate navigation techniques through the collected event stream. In our framework the visualization phase is handled by two modules, an Event Manager and a Visualization Manager. How these two modules interact is described below.

6.2.1 Visualizing Events

Within any system that presents program visualizations, some top-level abstraction should coordinate the different visualization tasks. In our framework, we refer to this entity as a Visualization Manager. The Visualization Manager acts as a manager for the individual visualizations (hereafter referred to as *views*) of the generated program events, and provides the user with a graphical interface for controlling the entire visualization. By communicating with both the user and the Event Manager, the events that the user wishes to be visualized can be retrieved from the Event Manager and passed on to the views.

Several informational models must be kept in order to translate the generated program events into useful views of an executing program. In order to recreate the program state as the visualization executes, a *program state model* must be maintained. This model should consist of the various entities within an object-oriented program: classes, their methods and attributes, global functions, instances, and the call stack. Events that describe the

static program structure (inheritance hierarchy, global functions) must be evaluated before execution events are received. Then, as the individual program events are processed, each event may affect the current program state.

Information to present the various views will be kept in *visualization models*. A visualization model is needed for each view of the executing program. These models should store information in addition to what is provided in the program state model—information that is required to maintain the state of each specific view. A consistent interface for the visualization models can be specified if the models are defined hierarchically as classes, with each individual visualization model as a sub-class of a generic visualization model base class. Such a hierarchy provides for structured inter-view communication and facilitates the addition of new views.

Corresponding to each visualization model would be a window displaying the particular view of that visualization. This hierarchy of visualization models and views is similar to that found in [DPHKV93], except that in our framework there is a one-to-one mapping from visualization model to view, while the IBM system can have a many-to-one mapping of models to views. Although it may seem that our structure loses flexibility, we feel that the overhead incurred in providing the many-to-one mapping may be too large for the animated type of views we wish to support.

In order to maintain the correspondence of the views to the actual program execution, the Visualization Manager must synchronize the update of each view. Additionally, if views need to communicate with one-another, the Visualization Manager will facilitate this as well. Inter-view communication is necessary to provide the user with contextual information during visualization of program events.

6.2.2 Managing Events

The Event Manager is essentially an event server for its client, the Visualization Manager. It is responsible for receiving and storing the generated program events. A natural storage facility for events might be an ordered queue, or even a specialized database that could support queries.

In our framework, the Event Manager is a separate entity from the Visualization Manager for two reasons. First, by having the Event Manager as a separate process, it can receive and store events while the Visualization Manager is visualizing previous events. We have parallelized the typically sequential task of receiving, decoding, and visualizing events.

Second, a separate Event Manager facilitates navigation through the event stream. During visualization, it should be possible to "fast-forward" or "rewind" events so that the user can focus quickly on past or future program execution states. In our framework the Visualization Manager does not have to ask for the next event, but can ask for the next instance of a particular event. This is especially useful when the user is only interested in specific events, such as object deletion.

The problem of supporting backtracking through the event stream can be likened to the problem of reverse execution[FB89]. In order to support what we call *reverse visualization* we need to 1) restore pre-existing program state models and visualization models, and 2) update the views to reflect the new models. The second part is fairly trivial based on our hierarchically defined visualization models; every visualization model simply provides an

"update" method.

Restoring previous states can be done by saving particular states, and then restoring them as the user rewinds the visualization. To provide incremental reversal at the event level, it would be necessary to save state after every event. This is clearly impractical. A more reasonable approach is to checkpoint particular events at some higher granularity. To implement reverse visualization, the most recent checkpointed state previous to the desired state can be restored, and in-between events replayed to reach the desired state. Because the Event Manager can store and retrieve events, such an implementation would be possible. In the past, event driven visualization systems typically have not allowed such backtracking, although programming environments such as PROVIDE [Moh88] have state-saving capabilities.

6.2.3 Views

As mentioned in Section 2, tools which support object-oriented program visualization will help in the following areas: understanding high-level aspects of a system, tracing dynamic message binding, analyzing dependencies between classes, locating system functionality, and resolving polymorphism[WH92]. Our visualization framework is intended to support views in each of the aforementioned problems, with the specific goal of handling real-world sized applications. Within this context, we can define several different *types* of views that can be created:

- Global views of program state and program element dependencies.
- Specific views of program elements.
- Statistical views of dependencies or actions existing during program execution.
- Textual views of program events and source code.

We envision that all four of these types of views will be useful, although discovering the content and presentation of useful views is a future question for our research.

7 Related Work

Several programming environments have been developed which include visualization capabilities. PECAN[Rei85] and PROVIDE[Moh88] are early program development systems that utilized graphical views such as data structure displays, a call-graph, and the call stack.

With respect to object-oriented systems in particular, more recent work includes Reiss' FIELD system[Rei90]. FIELD contains an extensive set of tools for developing and maintaining C++ programs. These tools include graphical aids such as class browsers[LMR92] and flow graphs, but the recent thrust of the work has been on 3-D display views[Rei93]. The BEE++[BGL93] object-oriented application framework supports dynamic analysis of distributed programs. It provides a platform for event monitoring, visualization, and graphical debugging. The analysis tools can be distributed across nodes, providing significant performance gains during visualization.

There has been a good deal of research on graphical presentation of object-oriented programs. Much of the research to date has focused on graphics used as a design aid in building object-oriented systems. The work of Booch[Boo91], Rumbaugh[RBP+91], *et. al.*,

Beck and Cunningham[BC89], Harel[Har88], Coleman, Hayes, and Bear[CHB92] all fits within this notion. The static diagramming techniques developed through this work can be quite helpful as a specification aid, but they are not appropriate for visualizing dynamic executions of programs, particularly those of very large programs.

Some work on visualizing executions has been conducted. Kleyn and Gingrich[KG88] sought to go beyond static displays by examining the dynamic behavior of object-oriented systems written in a Common Lisp-style language. Their GraphTrace tool illustrates structural and behavioral views of object-oriented systems by recording message traffic for subsequent replay. The tool's displays mainly involve graph diagrams consisting of nodes and arcs. Animation, however, is restricted to simply highlighting and annotating graph nodes. Böcker and Herczeg[BH90] provide more extensive animation of Smalltalk-80 traces with the Track system. Track allows programmers to visually specify message tracing as a debugging aid. At execution time, the system presents an animation of the messages sent between objects.

Two recent research thrusts are most similar to what we seek to accomplish in this work. De Pauw, Helm, Kimelman, and Vlissides at IBM have developed visualization techniques and a system for presenting attributes of object-oriented systems, more specifically, C++ programs[DPHKV93]. The authors developed instrumentation techniques that are portable and can extract the needed summary information about a program's execution. They also developed views, most of which are chart-like, that present summary information about the execution. Their views display instance creation and destruction, inter and intra-class calls, allocation histories, and so on. These views appear to be quite effective, and we would plan to adopt similar ones in our work. The information they capture, however, is mostly post-mortem summary information, whereas we seek to portray more of the run-time dynamics of a system. We also focus on providing effective navigation techniques between views and visualization events.

Koike is also exploring the use of 3D graphics to illustrate object-oriented systems[Koi93]. His techniques portray message traffic in a system with respect to the class hierarchy and the method list. This is accomplished by encoding the information within one 3D structure, which when viewed down the z-axis (the xy plane) presents the class hierarchy, and when viewed down the x-axis (the yz plane) presents the method list. Koike's technique is a novel way to encode much information within one view. We will explore similar techniques as Reiss' and Koike's, although our focus will primarily remain on 2D graphics.

8 Conclusions

It is clear that program development and maintenance are two of the most vital activities that occur in computing. Each relies on, and can benefit from, an increased level of *program understanding*. Current techniques for program comprehension often rely on some internal representation of a program from which useful information is derived. Our hypothesis is that *visualization* of object-oriented programs can be and will be an invaluable tool for object-oriented program understanding.

We have developed a visualization system for visualizing the execution of simple C++ programs. Based on this work, a framework for visualizing the execution of object-oriented programs has been defined. The framework provides a means for visualization with little

or no programmer intervention, and a mechanism which allows users to focus quickly by navigating through a collected stream of interesting events. Although this framework was developed to visualize object-oriented programs, its principles and structure could be applied to any event-driven visualization.

Our future research involves developing program views and implementing the framework itself. An empirical study will follow once a prototypical tool has been developed.

References

[BC89] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings of the ACM OOPSLA '89 Conference*, pages 1–6, New Orleans, LA, October 1989.

[BGL93] B. Bruegge, T. Gottschalk, and Bin Luo. A framework for dynamic program analyzers. In *SIGPLAN Notices: Proceedings of the 8th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 28, pages 65–82, Sept.-Oct. 1993.

[BH90] Heinz-Dieter Bocker and Jurgen Herczeg. What tracers are made of. In *Proceedings of the ECOOP/OOPSLA '90 Conference*, pages 89–99, Ottawa, Ontario, October 1990.

[BM82] V.R. Basili and H.D. Mills. Understanding and documenting programs. *IEEE Computer*, Oct 1982.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.

[Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–54, 1983.

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[DPHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the ACM OOPSLA '93 Conference*, pages 326–37, Washington, D.C., October 1993.

[DPKV94] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In *Proceedings of the European Conference on Object-Oriented Programming '94*, 1994.

[FB89] S.I. Feldman and C.B. Brown. IGOR: A system for program debugging via reversible execution. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24, pages 112–123, Jan 1989.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[KG88] Michael F. Kleyn and Paul C. Gingrich. GraphTrace - understanding object-oriented systems using concurrently animated views. In *Proceedings of the ACM OOPSLA '88 Conference*, pages 191–205, San Diego, CA, September 1988.

[Koi93] Hideki Koike. The role of another spatial dimension in software visualization. *ACM Transactions on Information Systems*, 11(3):266–286, July 1993.

[LA93] Panos E. Livadas and Scott D. Alden. A toolset for program understanding. In *Proceedings of the 2nd Workshop on Program Comprehension*, pages 110–118, Capri, Italy, July 8-9 1993.

[LAD+93] Panagiotis Linos, Philippe Aubet, Laurent Dumas, Yan Helleboid, Patricia Lejeune, and Phillippe Tulula. Facilitating the comprehension of c programs: An experimental study. In *Proceedings of the 2nd Workshop on Program Comprehension*, pages 55–63, Capri, Italy, July 8-9 1993.

[Let86] S. Letovsky. Cognitive processes in program comprehension. In E. Solloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79, Norwood, NJ, 1986.

[LMR92] Moises Lejter, Scott Meyers, and Steven Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992.

[LPLS86] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In E. Solloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 80–98, Norwood, NJ, 1986.

[Mey90] Scott Meyers. Working with object-oriented programs: The view from the trenches is not always pretty. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 51–65, September 1990.

[Moh88] Thomas G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–57, June 1988.

[PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

[Pen87] N. Pennington. Comprehension strategies in programming. In G.M. Olsen, S. Sheppard, and E. Solloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–13, Norwood, NJ, 1987.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New York, NY, 1991.

[Rei85] Steven P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–85, March 1985.

[Rei90] Steven P. Reiss. Interacting with the FIELD environment. *Software—Practice & Experience*, 20(S-1), June 1990.

[Rei93] Steven P. Reiss. A framework for abstract 3D visualization. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 108–115, Bergen, Norway, August 1993.

[SAE88] Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and processes in the comprehension of computer programs. In M. Chi, R. Glaser, and M. Farr, editors, *The Nature of Expertise*, pages 129–152. Lawrence Erlbaum Associates, 1988.

[SP92] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 3–10, Seattle, WA, September 1992.

[SS92] John J. Shilling and John T. Stasko. Using animation to design, document and trace object-oriented systems. Technical Report GIT-GVU-92-12, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, June 1992.

[TGP89] David Taenzer, Murthy Ganti, and Sunil Podar. Object-oriented software reuse: The yo-yo problem. *Journal of Object-Oriented Programming*, 2:30–35, Sept.-Oct. 1989.

[vMV93] A. von Mayrhauser and A.M. Vans. From program comprehension to tool requirements for an industrial environment. In *Proceedings of the 2nd Workshop on Program Comprehension*, pages 78–86, Capri, Italy, July 8-9 1993.

[WH92] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.

[YB93] E.J. Younger and K.H. Bennett. Model-based tools to record program understanding. In *Proceedings of the 2nd Workshop on Program Comprehension*, pages 87–95, Capri, Italy, July 8-9 1993.